Java Chapter 8: Strings

Contents

Time required: 90 minutes

Read: Think Java

1. Chapter 6 Loops and Strings

Do: Online Tutorial

Go through the following tutorials.

- Java Strings
- Java Concatenation
- Java Numbers and Strings
- Java Special Characters
- Wikibooks Java Strings

Java Primitive and Reference Data Types

Primitive types are the most basic data types available within the Java language. The eight primitives defined in Java are int, byte, short, long, float, double, boolean, and char – those aren't considered objects and represent raw values.

These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose — containing pure, simple values of a kind. Because these data types are defined into the Java type system by default, they come with several operations predefined. You cannot define a new operation for such primitive types. In the Java type system, there are three further categories of primitives:

- Numeric primitives: short, int, long, float and double. These primitive data types hold only numeric data. Operations associated with such data types are those of simple arithmetic (addition, subtraction, etc.) or of comparisons (is greater than, is equal to, etc.)
- Textual primitives: byte and char. These primitive data types hold characters (that can be Unicode alphabets or even numbers). Operations associated with such types are those of textual manipulation (comparing two words, joining characters to make words, etc.). However, byte and char can also support arithmetic operations.
- Boolean and null primitives: boolean and null.

All primitive types have a fixed size and are limited to a range of values.

A String in Java is a non-primitive data type because it refers to an object. A String variable holds a memory address. This memory address is a reference to the String object.

The following tutorial will help you understand the difference between primitive and reference variables in Java.

- https://java-programming.mooc.fi/part-5/3-primitive-and-reference-variables

Using String Objects

As we know, a Java program is a collection of interacting objects, where each object is a module that encapsulates a portion of the program's attributes and actions. Objects belong to classes, which serve as templates or blueprints for creating objects. Think again of the cookie cutter analogy.

A class is like a cookie cutter. Just as a cookie cutter is used to shape and create individual cookies, a class definition is used to shape and create individual objects.

Programming in Java is primarily a matter of designing and defining class definitions, which are then used to construct objects. The objects perform the program's desired actions. To push the cookie cutter analogy a little further, designing and defining a class is like building the cookie cutter. Obviously, very few of us would bake cookies if we first had to design and build the cookie cutters. We'd be better off using a pre-built cookie cutter. By the same token, rather than designing our own classes, it will be easier to get into "baking" programs if we begin by using some predefined Java classes.

Strings

Strings are very useful objects in Java and in all computer programs. They are used for inputting and outputting all types of data. Therefore, it essential that we learn how to create and use String objects.

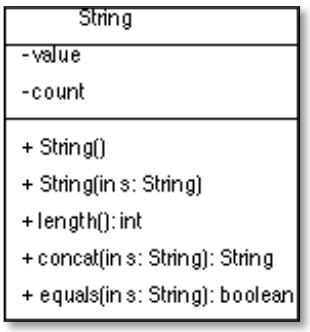


Figure 2.1 provides an overview of a very small part of Java's String class.

In addition to the two String() constructor methods, which are used to create strings, it lists several useful instance methods that can be used to manipulate strings. The String class also has two instance variables. One stores the String's value, which is a string of characters such as "Hello98", and the other stores the String's count, which is the number of characters in its string value.

To get things done in a program we send messages to objects. The messages must correspond to the object's instance methods. Sending a message to an object is a matter of calling one of its instance methods. In effect, we use an object's methods to get the object to perform certain actions for us. For example, if we have a String, named str and we want to find out how many characters it contains, we can call its length() method, using the

expression str.length(). If we want to print str's length, we can embed this expression in a print statement:

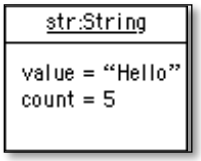
```
System.out.println (str.length( )); // Print str's length
```

In general, to use an object's instance method, we refer to the method in Dot notation dot notation by first naming the object and then the method:

objectName.methodName();

The objectName refers to a particular object, and the methodName() refers to one of its instance methods.

As this example makes clear, instance methods belong to objects, and to use a method, you must first have an object that has that method. To use one of the String methods in a program, we must first create a String object.



To create a String object in a program, we first declare a String variable/reference.

```
String str; // Declare a String variable named str
```

Figure 2.2: A String object stores a sequence of characters and a count giving the number of characters.

We create a String object by using the new keyword in conjunction with one of the String() constructors.

We assign the new object to the variable we declared:

```
str = new String ("Hello"); // Create a String object
```

This example will create a String that contains, as its value, the word "Hello" that is passed in by the constructor. The String object that this creates is shown in Figure 2.2.

We can also use a constructor with a parameter. Note that in this case we combine the variable declaration and the object creation into one statement:

```
String str = new String ("Hello"); // Create a String object
```

Strings are used so often that most programming languages have a shortcut method to create strings. This is Java's shortcut to create a String object.

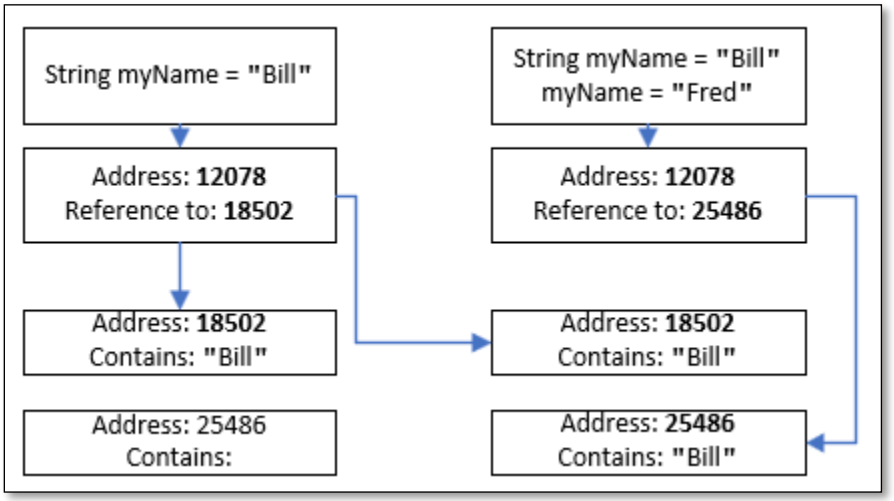
```
String str = "Hello"; // Create a String object
```

variables that are declared to be of a type equal to a class name are designed to store a reference to an object of that type. (A reference is also called a pointer because it points to

the memory address where the object itself is stored.) A constructor creates an object somewhere in memory and supplies a reference to it that is stored in the variable. For that reason, variables that are declared as a type equal to a class name are said to be variables of reference type or reference variables. Reference variables have a special default value called null after they are declared and before they are assigned a reference. It is possible to check whether a reference variable contains a reference to an actual object by checking whether or not it contains this null pointer.

Once you have constructed a String object, you can use any String method. As we already saw, we use dot notation to call one of the methods. Thus, we first mention the name of the object followed by a period (dot), followed by the name of the method. For example, the following statements print the lengths of our two strings:

```
System.out.println(str.length());
System.out.println(str2.length());
```



Characters

The **char** data type is a primitive data type which holds a single character. A char variable can be compared using relational operators. a == a

Java also has a Character class which contains methods for operating on characters. The Character class is a wrapper class. A wrapper class is a class that wraps(converts) a primitive datatype to an object.

Method	**Description**
`isLetter()`	Determines whether the specified char value is a letter.

`isDigit()`	Determines whether the specified char value is a digit (number).
`isLetterOrDigit()`	Determines whether the specified char value is a letter or a digit
`isWhitespace()`	Determines whether the specified char value is white space.
`isUpperCase()`	Determines whether the specified char value is uppercase.
`isLowerCase()`	Determines whether the specified char value is lowercase.
`toUpperCase()`	Returns the uppercase form of the specified char value.
`toLowerCase()`	Returns the lowercase form of the specified char value.

Tutorial 1

This tutorial shows examples of using Character methods.

```java
import java.util.Scanner;

public class CharacterClassDemo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        char userChar;

        // Get a character from the user
        System.out.print("Enter a single character: ");
        // The scanner class does not have a character method.
        // We get a string, then get the first character
        userChar = input.next().charAt(0);

        // Use the result directly in an if statement
        if (Character.isLetter(userChar)) {
            System.out.println(userChar + " is a letter.");
        } else {
            System.out.println(userChar + " is not a letter.");
        }

        // Store the result in a boolean variable
        boolean isDigitResult = Character.isDigit(userChar);
        if (isDigitResult == true) {
            System.out.println("The character '" + userChar + "' is a digit. ");
        } else {
            System.out.println("The character '" + userChar + "' is not a digit.");
        }

        boolean isLetterOrDigitResult = Character.isLetterOrDigit(userChar);
        System.out.println(userChar + " is a letter or a digit: "
                + isLetterOrDigitResult);

        if (Character.isUpperCase(userChar)) {
            System.out.println(userChar + " is uppercase");
        } else {
            System.out.println(userChar + " is not uppercase");
        }

        System.out.println(userChar + " converted to upper case is: "
                + Character.toUpperCase(userChar));
        System.out.println(userChar + " converted to lower case is: "
                + Character.toLowerCase(userChar));

        // Close the open operating system resource
        input.close();
    }
}
```

Example runs:

```
Enter a single character: b
b is a letter.
The character 'b' is not a digit.
b is a letter or a digit: true
b is not uppercase
b converted to upper case is: B
b converted to lower case is: b
```

```
Enter a single character: 4
4 is not a letter.
The character '4' is a digit.
4 is a letter or a digit: true
4 is not uppercase
4 converted to upper case is: 4
4 converted to lower case is: 4
```

String Methods

String is a "class" and as such has methods that are properties of the class. To use a method, we need the dot operator.

length()

Returns the length of a string as an integer value.

Example:

```
// Assigns the length of a city name to an int
int cityLength = city.length();
System.out.println(cityLength)
```

equals( )

Used to compare two strings with the method to see if they are exactly the same, this includes any blanks or spaces within the string.

Example:

```
// Check to see if the user entered a dog name of Snoopy
if (dogname.equals("Snoopy");
System.out.println("The user entered Snoopy.”);
```

Tutorials

- Java String equals() Method

charAt(n)

Returns a char at the n location of the string. Subscripting a String (just like other iterables) starts at 0.

Example:

```
String holiday="Thanksgiving";
System.out.println(holiday.charAt(4));
>> k
String puppy ="Wally";
System.out.println(puppy.charAt(0));
>> W
```

Tutorials

- Java String charAt() Method

toUpperCase(s) or toLowerCase(s)

Returns a String with all UPPER CASE or all lower case.

Example:

```
String cityname = "Houston";
System.out.println(cityname.toLowerCase());
>> houston
System.out.println(cityname.toUpperCase());
>> HOUSTON
```

Tutorials

- Java String toUpperCase() Method
- Java String toLowerCase() Method

trim()

This is a value returning method that returns a copy of the argument after removing its leading and trailing blanks (not embedded blanks - blanks within the statement).

Example:

```
String burp = "          hic up          ";
System.out.println(burp.trim());
>> hic up
```

Tutorials

- Java String trim() Method

substring(Start, End)

This is a value returning method. A string is returned beginning at Start subscript up to but not including the End subscript.

Example:

```
String sport = "football";
System.out.println(sport.substring(1,3));
>> oo
System.out.println(sport.substring(1));
>> ootball
```

indexOf(c)

This method returns the position of the first occurrence of specified character(c) in a string.

Example:

```
// Find the first comma in a city, state, zip listing
String city = "Fulton, New York 13069";
System.out.println(city.indexOf(","));
>> 6
```

Tutorials

- Java String indexOf() Method

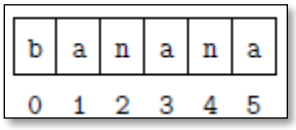
CharAt()

Some of the most interesting problems in computer science involve searching and manipulating text. In the next few sections, we'll discuss how to apply loops to strings. Although the examples are short, the techniques work the same whether you have one word or one million words.

Strings provide a method named charAt(). It returns a char, a data type that stores an individual character (as opposed to strings of them):

```
String fruit = "banana";
char letter = fruit.charAt(0);
```

The argument 0 means that we want the character at index 0. String indexes range from 0 to n - 1, where n is the length of the string. The character assigned to letter is 'b':



Characters work like the other data types you have seen. You can compare them using relational operators:

```
if (letter == 'A') {
System.out.println("It's an A!");
}
```

Character literals, like 'A', appear in single quotes. Unlike string literals, which appear in double quotes, character literals can contain only a single character. Escape sequences, like '\t', are legal because they represent a single character.

The increment and decrement operators also work with characters. This loop displays the letters of the alphabet:

```
System.out.print("Roman alphabet: ");
for (char c = 'A'; c <= 'Z'; c++) {
System.out.print(c);
}
System.out.println();
```

The output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Java uses Unicode to represent characters, so strings can store text in other alphabets like Cyrillic and Greek, and non-alphabetic languages like Chinese. You can read more about it at the Unicode website (https://unicode.org).

In Unicode, each character is represented by a \code point", which you can think of as an integer. The code points for uppercase Greek letters run from 913 to 937, so we can display the Greek alphabet like this:

```
System.out.print("Greek alphabet: ");
for (int i = 913; i <= 937; i++) {
    System.out.print((char) i);
}
System.out.println();
```

This example uses a char type cast to convert each integer (in the range) to the corresponding character. Try running the code and see what happens.

Which Loop to Use?

for and while loops have the same capabilities; any for loop can be rewritten as a while loop, and vice versa. For example, we could have printed letters of the alphabet by using a while loop:

```
System.out.print("Roman alphabet: ");
char c = 'A';
while (c <= 'Z') {
System.out.print(c);
c++;
}
System.out.println();
```

You might wonder when to use one or the other. It depends on whether you know how many times the loop will repeat.

A for loop is "definite", which means we know, at the beginning of the loop, how many times it will repeat. In the alphabet example, we know it will run 26 times. In that case, it's better to use a for loop, which puts all of the loop control code on one line.

A while loop is "indefinite", which means we don't know how many times it will repeat. For example, when validating user input, it's impossible to know how many times the user will enter a wrong value. In this case, a while loop is more appropriate.

String Iteration

Strings and Arrays are very similar in how you work with them. They are both reference types and have indexes and class methods.

Strings provide a method called length that returns the number of characters in the string. The following loop iterates the characters in fruit and displays them, one on each line:

```
for (int i = 0; i < fruit.length(); i++) {
    char letter = fruit.charAt(i);
    System.out.println(letter);
}
```

Because length is a method, you have to invoke it with parentheses (there are no arguments). When i is equal to the length of the string, the condition becomes false and the loop terminates.

To find the last letter of a string, you might be tempted to do something like the following:

```
int length = fruit.length();
char last = fruit.charAt(length); // wrong!
```

This code compiles and runs, but invoking the charAt method throws a StringIndexOutOfBoundsException. The problem is that there is no sixth letter in "banana". Since we started counting at 0, the six letters are indexed from 0 to 5. To get the last character, subtract 1 from length:

```
int length = fruit.length();
char last = fruit.charAt(length - 1); // correct
```

Many string algorithms involve reading one string and building another. For example, to reverse a string, we can concatenate one character at a time:

```
public static String reverse(String s) {
    String r = "";
    for (int i = s.length() - 1; i >= 0; i--) {
        r += s.charAt(i);
    }
    return r;
}
```

The initial value of r is "", which is an empty string. The loop iterates the indexes of s in reverse order. Each time through the loop, the += operator appends the next character to r. When the loop exits, r contains the characters from s in reverse order. So the result of reverse("banana") is "ananab".

The indexOf() Method

To search for a special character in a string, you could write a for loop and use charAt as in the previous section. However, the String class already provides a method for doing just that:

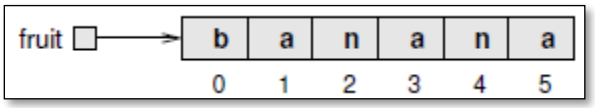
```
String fruit = "banana";
int index = fruit.indexOf('a'); // returns 1
```

This example finds the index of 'a' in the string. But the letter appears three times, so it's not obvious what indexOf() might do. According to the documentation, it returns the index of the first appearance.

To find subsequent appearances, you can use another version of indexOf, which takes a second argument that indicates where in the string to start looking:

```
int index = fruit.indexOf('a', 2); // returns 3
```

To visualize how indexOf and other String methods work, it helps to draw a picture like the one below. The previous code starts at index 2 (the first 'n') and finds the next 'a', which is at index 3.



If the character happens to appear at the starting index, the starting index is the answer. So fruit.indexOf('a', 5) returns 5. If the character does not appear in the string, indexOf returns -1. Since indexes cannot be negative, this value indicates the character was not found.

You can also use indexOf to search for an entire string, not just a single character. For example, the expression fruit.indexOf("nan") returns 2.

Substrings

In addition to searching strings, we often need to extract parts of strings. The substring method returns a new string that copies letters from an existing string, given a pair of indexes:

- fruit.substring(0, 3) returns "ban"
- fruit.substring(2, 5) returns "nan"
- fruit.substring(6, 6) returns ""

Notice that the character indicated by the second index is not included. Defining substring this way simplifies some common operations. For example, to select a substring with length len, starting at index i, you could write

fruit.substring(i, i + len)

Like most string methods, substring is overloaded. That is, there are other versions of substring that have different parameters. If it's invoked with one argument, it returns the letters from that index to the end:

- fruit.substring(0) returns "banana"
- fruit.substring(2) returns "nana"
- fruit.substring(6) returns ""

The first example returns a copy of the entire string. The second example returns all but the first two characters. As the last example shows, substring returns the empty string if the argument is the length of the string.

We could also use fruit.substring(2, fruit.length() - 1) to get the result "nana". But calling substring with one argument is more convenient when you want the end of the string.

String Comparison

When comparing strings, it might be tempting to use the == and != operators.

The problem is that the == operator checks whether the two operands refer to the same object. Even if the answer is "yes", it will refer to a different object in memory than the literal string "yes" in the code.

The correct way to compare strings is with the equals method, like this:

```
if (answer.equals("yes")) {
    System.out.println("Let's go!");
}
```

This example invokes equals on answer and passes "yes" as an argument. The equals method returns true if the strings contain the same characters; otherwise, it returns false.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.